

CRUD Operation in FastAPI - Sandwich Maker Machine Program

In this assignment, you will have the opportunity to explore and implement the fundamental principles of creating, reading, updating, and deleting (CRUD) operations on a database using FastAPI, a modern and highly efficient web framework for building APIs with Python.

The primary goal of this assignment is to develop a comprehensive understanding of CRUD operations by implementing them on a database. Specifically, you will be working with an extended version of a Sandwich Maker Machine Program. Throughout this assignment, you will become familiar with the following key concepts and techniques:

- **Modular Implementation with FastAPI:** You will learn how to structure your code in a modular fashion using FastAPI, ensuring that your application is organized, maintainable, and extensible.
- **Database Connectivity:** You will explore how to establish a connection to a database from within a FastAPI application. This connection will serve as the foundation for storing and retrieving data related to our Sandwich Maker Machine Program.
- **Table Creation with FastAPI:** You will dive into the process of defining database tables within your FastAPI application, specifying the schema, and handling migrations if necessary.
- **Endpoint Creation:** You will create API endpoints for performing CRUD operations on the database. These endpoints will allow you to interact with the Sandwich Maker Machine Program by adding, retrieving, updating, and deleting sandwich-related data.

Initial Setup:

1- Source Code:

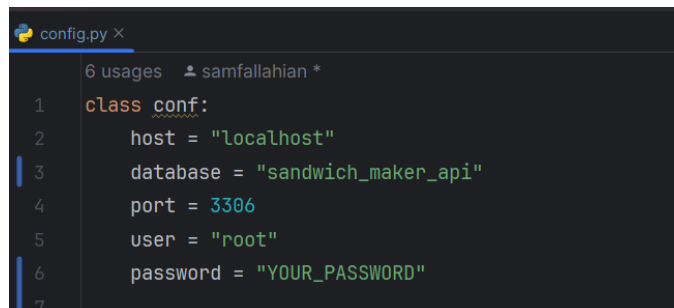
On Canvas, you can find the [code base](#) on the assignment page.

2- Open your assignment project and install the following packages in the virtual environment using `pip` command.

```
pip install fastapi
pip install "uvicorn[standard]"
pip install sqlalchemy
pip install pymysql
```

3- Set database credential:

- Create a brand-new database and name it "sandwich_maker_api".
- Open [api/dependencies/config.py](#) and set database name, MySQL username and password.



```
config.py x
6 usages  samfallahian *
1 class conf:
2     host = "localhost"
3     database = "sandwich_maker_api"
4     port = 3306
5     user = "root"
6     password = "YOUR_PASSWORD"
7
```

4- Run the program:

- First go to the assignment folder the run the following command
`uvicorn api.main:app --reload`
- If you check your database again, you'll see that the sqlalchemy package has automatically generated five new tables.

Project Structure and Architecture:

Our FastAPI project is organized like a well-structured team, with each folder and file having a specific role. At the root of our project, we have the `app` folder, which is the heart of our FastAPI application. Let's explore its contents:

Controllers Package: Inside the `controllers` package folder, you'll find Python files. These files are like the conductors of our orchestra. Each of them handles CRUD (Create, Read, Update, Delete) functions for a specific part of our application. Think of them as the experts who know how to talk to the database. Each database table will have its separate `.py` file in this package.

Models Package: Inside the `models` package folder, we have two crucial files:

- `models.py`: This file contains Python classes that describe the structure of our database tables. It tells us what each table looks like, what kind of data it stores, and how different tables are related. SQLAlchemy, a powerful database toolkit, uses this file to create tables in our database.
- `schemas.py`: Think of this file as a blueprint for the data structures that our API will use. It defines how the data should look when we send or receive it through our API. These schemas match our data models, ensuring consistency.

Dependencies Package: Within the `dependencies` package, you'll find two Python files:

- `config.py`: This file is like our project's settings manager. It stores constant configuration details that our entire project might need, such as database credentials. It keeps things organized and easily accessible.
- `database.py`: This file is your database connection maestro. It's responsible for establishing connections to the database and handling all the interactions with it. Whenever our application needs to talk to the database, this file makes it happen.

main.py : Finally, we have the `main.py` file located in the root of the `api` folder. It's the entry point of our FastAPI application. Think of it as the receptionist at a hotel. It receives API requests, builds API endpoints, and knows which controller to send the request to. It's the central hub that directs traffic in our application.

Understanding this project structure is crucial as it helps us keep our code organized, maintainable, and efficient.

Task:

In our FastAPI project, we have a database with five tables: `sandwiches`, `resources`, `recipes`, `orders`, and `order_details`. We've already made progress by implementing everything related to the database and models and full functions and endpoints for the orders table. Now, it's time to extend this functionality to the remaining tables. Here's a systematic approach:

- 1- **Select the Next Table:** Begin by selecting the next table you want to implement. You can choose any of the tables: `sandwiches`, `resources`, `recipes`, `orders`, or `order_details`. For this task, let's assume you've chosen the sandwiches table.
- 2- **Create a New controller:** Inside the `controllers` package folder, create a new Python file with the same name as the chosen table. In this case, name the file `sandwiches.py`. This file will be dedicated to handling CRUD operations for the sandwiches table.
- 3- **Follow the Sample (orders table):** Now, open the newly created `sandwiches.py` file and follow the pattern set by the sample file (`orders.py`). Implement the following functions for CRUD operations:
 - `create`: This function allows the creation of new data in the table.
 - `update`: This function handles updates to existing data.
 - `delete`: Implement a function to delete data from the table.
 - `read_all`: Develop a function to retrieve all rows in table.
 - `read_one`: Create a function to retrieve a specific data by its id.
- 4- **Modify main.py:** Open the main.py file located in the root of the `api` folder. Here, you will add corresponding endpoints for the new table you're working on. Make sure to follow the pattern set by the sample (`orders`) for creating API endpoints. Don't forget to change the endpoint `path` and `tags` to match the new table. This is crucial for keeping your API well-organized.
- 5- **Repeat for All Tables:** Repeat this process for each of the remaining tables (`resources`, `recipes`, `order_details`). Select a table, create a controller file, implement CRUD functions, and add corresponding endpoints in `main.py`.
- 6- **Use Interactive API docs to send requests and make sure that your code is functioning properly.**

<http://127.0.0.1:8000/docs>

Click the 'Try it out' button, enter an arbitrary value, and send requests.

By following this methodical approach, you'll systematically build CRUD functionality for all the tables in your FastAPI project. This step-by-step process ensures that you maintain consistency and organization in your project while gradually extending its functionality to cover all the required tables.

Code Anatomy:

Based on sample implementation (`orders`), in this section, we'll understand the inner workings of our codebase for crud operations. We'll break down the key elements, from route annotations and decorators to controller functions, we'll explore the anatomy of our code to grasp how it handles HTTP requests, interacts with the database, and responds to client requests.

Controller:

```
def create(db: Session, order):
    db_order = models.Order(
        customer_name=order.customer_name,
        description=order.description
    )
    db.add(db_order)
```

```
db.commit()
db.refresh(db_order)
return db_order
```

1. `def create(db: Session, order):` This line defines a Python function named `create` that takes two arguments:
 - o `db`: A database session object (likely provided by SQLAlchemy's `Session` class).
 - o `order`: An object that represents the data to be inserted into the database. The data structure of this object should match the expected fields of the `orders` table in the database.
2. `db_order = models.Order(...)`: This section creates a new instance of the `Order` model from the `models` module. It assigns the values from the `order` object (such as `customer_name` and `description`) to the corresponding fields of the `db_order` object.
3. `db.add(db_order)`: Here, the `db_order` object is added to the database session using the `add` method. This action stages the object for insertion into the database.
4. `db.commit()`: This line commits the changes made to the database session to persist the new `db_order` object into the database.
5. `db.refresh(db_order)`: After committing the changes, this step refreshes the `db_order` object. It ensures that the `db_order` object now reflects the data as it exists in the database, including any auto-generated fields or default values.
6. `return db_order`: Finally, the function returns the newly created `db_order` object. This allows the caller of the function to access the database-generated values or perform further operations if needed.

```
def read_all(db: Session):
    return db.query(models.Order).all()
```

1. `def read_all(db: Session):` This line defines a Python function named `read_all`. It takes a single argument, `db`, which is a database session object (presumably provided by SQLAlchemy's `Session` class).
2. `return db.query(models.Order).all()`: In this line, the function performs the following steps:
 - o `db.query(models.Order)`: This part initiates a database query using the provided session `db`. It specifies that we want to query the `Order` model (which presumably maps to the `orders` table in the database).
 - o `.all()`: This method is called on the query object to retrieve all records from the `orders` table. It returns a list of `Order` objects (or an empty list if there are no records).

So, when you call the `read_all` function and pass a database session to it, it executes a query to fetch all records from the `orders` table and returns them as a list of `Order` objects.

```
def read_one(db: Session, order_id):
    return db.query(models.Order).filter(models.Order.id == order_id).first()
```

1. `def read_one(db: Session, order_id):` This line defines a Python function named `read_one`. It takes two arguments:
 - o `db`: A database session object (presumably provided by SQLAlchemy's `Session` class).
 - o `order_id`: An identifier that is used to locate the specific order you want to retrieve from the database.
2. `return db.query(models.Order).filter(models.Order.id == order_id).first()`: In this line, the function performs the following steps:
 - o `db.query(models.Order)`: This part initiates a database query using the provided session `db`. It specifies that we want to query the `Order` model (which maps to the `orders` table in the database).
 - o `.filter(models.Order.id == order_id)`: The `filter` method is used to specify a condition that filters the records in the query. In this case, it filters the records to find the one where the `id` column matches the provided `order_id`.

- `.first()`: This method is called on the query object to retrieve the first record that matches the filtering condition. Since we are looking for a single record based on the `order_id`, `.first()` returns that record (or `None` if no matching record is found).

So, when you call the `read_one` function and pass a database session and an `order_id` to it, it executes a query to find and return the specific order record from the orders table in the database based on the provided `order_id`. This function is useful when you want to retrieve the details of a particular order using its unique identifier.

```
def update(db: Session, order_id, order):
    db_order = db.query(models.Order).filter(models.Order.id == order_id)
    update_data = order.dict(exclude_unset=True)
    db_order.update(update_data, synchronize_session=False)
    db.commit()
    return db_order.first()
```

1. `def update(db: Session, order_id, order)::` This line defines a Python function named `update`. It takes three arguments:
 - `db`: A database session object (presumably provided by SQLAlchemy's `Session` class).
 - `order_id`: An identifier that is used to locate the specific order you want to update in the database.
 - `order`: An object that represents the updated data for the order.
2. `db_order = db.query(models.Order).filter(models.Order.id == order_id):` This line initiates a database query using the provided session `db`. It filters the query to select the specific order with an `id` matching the provided `order_id`. The `db_order` variable holds the query result.
3. `update_data = order.dict(exclude_unset=True):` Here, the code extracts the update data from the provided order object. It uses the `.dict()` method to convert the order object into a dictionary, excluding any unset (undefined) fields. This dictionary will be used to update the database record.
4. `db_order.update(update_data, synchronize_session=False):` The update method is called on the `db_order` query object. It updates the selected database record with the data from the `update_data` dictionary. The `synchronize_session=False` parameter ensures that the session is not synchronized immediately, which can be more efficient for updates.
5. `db.commit():` This line commits the changes made to the database, effectively saving the updated record.
6. `return db_order.first():` Finally, the function returns the updated order record as it exists in the database. The `db_order.first()` call retrieves the first (and only) record that matches the filtering condition.

In summary, this function allows you to update a specific order record in the orders table by providing the `order_id` and an order object with the updated data.

```
def delete(db: Session, order_id):
    db_order = db.query(models.Order).filter(models.Order.id == order_id)
    db_order.delete(synchronize_session=False)
    db.commit()
    return Response(status_code=status.HTTP_204_NO_CONTENT)
```

1. `def delete(db: Session, order_id)::` This line defines a Python function named `delete`. It takes two arguments:
 - `db`: A database session object.
 - `order_id`: An identifier that is used to locate the specific order you want to delete in the database.
2. `db_order = db.query(models.Order).filter(models.Order.id == order_id):` This line initiates a database query using the provided session `db`. It filters the query to select the specific order with an `id` matching the provided `order_id`. The `db_order` variable holds the query result.

3. `db_order.delete(synchronize_session=False)`: The `delete` method is called on the `db_order` query object. It deletes the selected database record without synchronizing the session. This means the session is not immediately updated to reflect the deletion, which can be more efficient for delete operations.
4. `db.commit()`: This line commits the changes made to the database, effectively deleting the selected record.
5. `return Response(status_code=status.HTTP_204_NO_CONTENT)`: Finally, the function returns a FastAPI Response object with a status code of 204 No Content. This status code indicates a successful deletion, and it's a common convention for RESTful APIs to return this status code after a deletion operation.

In summary, this function allows you to delete a specific order record from the order table by providing the `order_id`. It performs the deletion, commits the changes to the database, and responds with a status code to indicate the success of the operation.

Main:

```
@app.post("/orders/", response_model=schemas.Order, tags=["Orders"])
def create_order(order: schemas.OrderCreate, db: Session = Depends(get_db)):
    return orders.create(db=db, order=order)

@app.get("/orders/", response_model=list[schemas.Order], tags=["Orders"])
def read_orders(db: Session = Depends(get_db)):
    return orders.read_all(db)

@app.get("/orders/{order_id}", response_model=schemas.Order, tags=["Orders"])
def read_one_order(order_id: int, db: Session = Depends(get_db)):
    order = orders.read_one(db, order_id=order_id)
    if order is None:
        raise HTTPException(status_code=404, detail="User not found")
    return order

@app.put("/orders/{order_id}", response_model=schemas.Order, tags=["Orders"])
def update_one_order(order_id: int, order: schemas.OrderUpdate, db: Session = Depends(get_db)):
    order_db = orders.read_one(db, order_id=order_id)
    if order_db is None:
        raise HTTPException(status_code=404, detail="User not found")
    return orders.update(db=db, order=order, order_id=order_id)

@app.delete("/orders/{order_id}", tags=["Orders"])
def delete_one_order(order_id: int, db: Session = Depends(get_db)):
    order = orders.read_one(db, order_id=order_id)
    if order is None:
        raise HTTPException(status_code=404, detail="User not found")
    return orders.delete(db=db, order_id=order_id)
```

1. `create_order`: Handles POST requests to create a new order, returning the created order data.
2. `read_orders`: Handles GET requests to retrieve a list of all orders, returning a list of order data.
3. `read_one_order`: Handles GET requests with a specific `order_id` to retrieve a single order, returning the order data.
4. `update_one_order`: Handles PUT requests with a specific `order_id` to update an order, returning the updated order data.

5. `delete_one_order`: Handles DELETE requests with a specific `order_id` to delete an order, returning a success status code (204 No Content).

Decorators:

- `@app.post`, `@app.get`, `@app.put`, `@app.delete`: These decorators specify the HTTP method (POST, GET, PUT, DELETE) that the following function handles.
- `"/orders/"` or `"/orders/{order_id}"`: These are the URL paths where the route is accessible. The curly braces `{ }` indicate dynamic path parameters.
- `response_model=schemas.Order`: Specifies the expected response model for the route, ensuring the API response conforms to the defined data structure.
- `tags=["Orders"]`: Tags the route for organizational and documentation purposes, helping categorize and find routes in API documentation.

These decorators define how each route handles incoming requests and the expected response structure. The functions connect these routes to the corresponding controller functions for data processing and database interaction.